ARCHI2 - Compte-rendu du TME5

Nicolas Phan

pour le 24 Mars 2018

Table des matières

1 Architecture matérielle

1.1 Question B1

Signification des arguments du constructer de PibusFrameBuffer :

- `name` : le nom du composant
- `tgtid` : Le target id, un nombre unique associéj à chaque composant et permettant de différencier chacun des composants connectés au bus.
- `segtab` : Une référence vers un objet de type `PibusSegmentTable` contenant la table des segments de l'espace adressable.
- `latency` : Le nombre de cycles pendant lesquels l'automate du controlleur du FBF restera à l'état `READ_WAIT` ou `WRITE_WAIT`. Cela représente la latence du périphérique en lecture/écriture.
- `width` : Le nombre de colonnes de pixels de l'ecran graphique controllé par le périphérique FBF.
- `height` : Idem pour le nombre de lignes de pixels.
- `subsampling` : Le format de pixel, autrement dit la manière dont il faut lire un octet des tampons du FBF pour en déduire la couleur du pixel correspondant.

1.2 Question B2

Le composant FBF ne traitera ici que des images en niveau de gris, donc ici seul le tableau de luminances est utile, il n'y aura besoin d'adresse que ce tableau, et il comporte 64 Ko (256 lignes de pixels $\times$ 256 colonnes de pixels $\times$ 1 octet par pixel).

La longueur du segment associé à de composant est donc de 64Ko.

2 Compilation de l'application logicielle

2.1 Question C1

Pour écrire dans le FB, il faut accéder aux registres adressables du controlleur du FB, qui sont dans une partie de l'espace adressable réservée au noyau. Le programme utilisateur doit donc demander au noyau (via un appel système) l'afficher ce qu'il désire à l'écran. Si un programme décide de contourner l'OS en écrivant directement dans les registres du FB, une exception de type segfault sera levée et l'OS réagira en terminant l'application fautive.

2.2 Question C2

Si une application écrivait pixel par pixel, alors l'écriture de l'image entière nécessiterait 64000 appels système, or un appel système est couteux en cycles d'horloge, beaucoup de temps d'exécution serait perdu par des va-et-viens entre le mode utilisateur et le mode kernel.

A l'inverse, si l'application construisait un tableau de 64 Ko et écrivait l'image en un seul appel système, le temps d'exécution serait réduit mais l'empreinte mémoire de l'application serait plus important.

Ainsi, afficher une image par 256 appel système écrivant chacun 256 pixels est le compromis entre rapidité d'exécution et empreinte mémoire réduite.

2.3 Question C3

`fb_sync_write` est une fonction permettant à l'application d'écrire dans le tableau du Frame Buffer. Ses arguments sont :

- `offset` est l'index de la case du tableau destination à partir de laquelle la fonction va écrire.
- `buffer` est un pointeur vers la première case du tableau contenant les données à écrire.
- `length` est le nombre de cases que l'on veut écrire.

3 Caractérisation de l'application logicielle

3.1 Question D1

L'affichage de l'image se fait par 4046323 instructions effectuées en 5012208 cycles. Ce qui donne un CPI de $\frac{5012208}{4046323} = 1.23871$, ce qui correspond au CPI indiqué dans l'affichage des statistiques.

3.2 Question D2

$$\begin{aligned}
\text{écriture} &= \texttt{WRITE_RATE} && = 0.141599 \\
\text{lecture} &= \texttt{UNCACHED_READ_RATE} + \texttt{CACHED_READ_RATE} && = 0.00130143 + 0.261667 = 0262969 \\
\text{taux de miss I} &= \texttt{IMISS_RATE} && = 2.05125.10^{-5} \\
\text{taux de miss D} &= \texttt{DMISS_RATE} && = 2.45564.10^{-5} \\
\text{cout d'un miss I} &= \texttt{IMISS_COST} && = 16.2 \\
\text{cout d'un miss D} &= \texttt{DMISS_COST} && = 15.2 \\
\text{cout d'écriture} &= \texttt{WRITE_COST} && = 0
\end{aligned}$$

Ces coûts sont des moyennes d'une grandeur qui varie au cours du temps, donc même si la grandeur en question a toujours une valeur entière, la moyenne, en revanche, ne l'est pas forcément.

3.3 Question D3

D'après l'affichage statistique :

$$\boxed{\text{nombre de transactions total} = 578331}$$

Ensuite, le nombre de transactions de chaque type est :

$$\begin{aligned}
\text{nombre de transactions IMISS} &= \text{nombre de MISS instruction} \\
&= \texttt{IMISS_RATE} \times \texttt{NB_INSTRUCTIONS} \\
&= 2.05125.10^{-5} \times 4046323 \\
&= 83
\end{aligned}$$

$$\boxed{\text{nombre de transactions IMISS} = 83}$$

$$\begin{aligned}
\text{nombre de transactions DMISS} &= \text{nombre de MISS données} \\
&= \texttt{DMISS_RATE} \times \texttt{NB_INSTRUCTIONS} \\
&= 2.45564.10^{-5} \times 4046323 \\
&= 99
\end{aligned}$$

$$\boxed{\text{nombre de transactions DMISS} = 99}$$

$$\begin{aligned}
\text{nombre de transactions UNC} &= \texttt{UNC_RATE} \times \texttt{NB_INSTRUCTIONS} \\
&= 0.00130143 \times 4046323 \\
&= 5266
\end{aligned}$$

$$\boxed{\text{nombre de transactions UNC} = 5266}$$

$$\begin{aligned}\text{nombre de transactions WRITE} &= \texttt{WRITE\_RATE} \times \texttt{NB\_INSTRUCTIONS} \\ &= 0.141599 \times 4046323 \\ &= 572955\end{aligned}$$

$$\boxed{\text{nombre de transactions WRITE} = 572955}$$

Enfin, lorsqu'on additionne le nombre de transactions pour chacun des types, on obtient $83 + 99 + 5266 + 572955 = 578403 \simeq 578331$, à quelques erreurs d'arrondi près, cela correspond bien au nombre total de transactions effectuées. Nous pouvons observer que la majorité des transactions effectuées sont des écritures.

4 Exécution sur l'architecture multi-processeurs

4.1 Question E1

Adaptation du code de `main.c` pour exécution sur un système multiprocesseurs :

```
    unsigned char   buf[NPIXEL];
    int             n = procid();
    int             nprocs = NB_PROCS;
    unsigned int    line;
    unsigned int    pixel;

    for(line = n ; line < NLINE ; line += nprocs)
    {
```

4.2 Question E2

La pile d'exécution contient les variables locales et paramètres passés lors d'appels de fonctions, comme ces données dont propres à chaque processus, il est indispensable que chacun des processus ait sa propre pile d'exécution.

Adaptation du code de `reset.s` pour exécution sur un système multiprocesseurs :

Pour l'adaptation, nous allons partitionner le segment de pile en sous-segments de 64Ko chacun, et le processeur n aura le sous-segment n. Ainsi, l'adresse de base du segment de pile du processeur n sera

$$\text{base\_n} = \texttt{SEG\_STACK\_BASE} + n \times 64\text{Ko}$$

Et l'adresse initiale du stack pointeur sera de :

$$\begin{aligned}\text{SP\_n} &= \texttt{SEG\_STACK\_BASE} + n \times 64\text{Ko} + 64\text{Ko} \\ \text{SP\_n} &= \texttt{SEG\_STACK\_BASE} + (n+1) \times 64\text{Ko}\end{aligned}$$

D'où le code assembleur suivant :

```
# initializes stack pointer
    la      $29,    seg_stack_base
    mfc0    $28,    $15,    1       # $28 = procid
    addiu   $28,    $28,    1       # $28 = procid+1
    sll     $28,    $28,    16      # $28 = (procid+1) * 64 Ko
    addu    $29,    $29,    $28     # $29 = base + (procid+1) * 64Ko
```

4.3 Question E3

Changer le nombre de processeurs implique de changer `NB_PROCS` dans `config.h`, autrement dit changer le code source du logiciel, il est donc nécessaire de recompiler le code après cela.

4.4 Question E4

	1 proc	2 proc	4 proc	6 proc	8 proc
cycles	5012470	2507508	1282629	1176653	1164973
speedup	1	2	4	4.2	4.3

TABLE 1 – Speedup VS nombre de processeurs

Jusqu'à $N = 4$, le speedup est linéaire comme prévu, mais à partir de $N = 6$, le speedup augmente peu car on arrive à saturation des capacités de transfert du bus. Comme il y a trop de processeurs demandeurs du bus, ce-dernier sature donc même s'il y a encore plus de processeurs, les processeurs doivent attendre plus longtemps avant d'obtenir l'accès au bus et y effectuer leurs écritures, donc en fin de compte le speedup stagne à 4.2.

5 Evaluation des temps d'accès au bus

5.1 Question F1

Les statistiques que l'on cherche à obtenir découlent du nombre de cycles de gel, de requêtes, d'instructions etc. durant toute la durée nécessaire pour afficher l'échiquier, **ni plus, ni moins**.

Après le `exit()`, le syystème d'exploitation continue de tourner et donc des instructions assembleur continuent de s'exécuter. Prendre une mesure de statistiques après le `exit()` c'est prendre en compte ces instructions post-exit() non pertinentes qui fausseraient les mesures.

C'est pour cela qu'il est important de prendre la mesure au moment où l'application se termine.

5.2 Question F2

NPROCS	1	2	4	6	8
cycles	5012470	2507508	1282629	1176653	1164973
IMISS_COST	15.9	19	32.7	56	78
DMISS_COST	15.4	18.8	56.2	100	136
WRITE_COST	0	0	0.15	3.2	7
ACCESS_TIME	1	1.1	3.2	8.7	12.8
CPI	1.24	1.24	1.27	1.7	2.3

TABLE 2 – Performances VS nombre de processeurs

5.3 Question F3

Au fur et à mesure que le nombre de processeurs augmente, nous observons que le coût des MISS et des WRITE, autrement dit le coût des accès au bus en général, augmente jusqu'à un facteur 10 de 1 processeur à 8 processeurs.

Si on calcule le nombre de cycles de gel dûs aux MISS :

$$\text{miss\_freeze\_cycles} = \text{miss\_cost} \times \text{miss\_rate} \times \text{nb\_instructions} = 10^2 \times 10^{-4} \times \text{nb\_instructions} = 10^{-2}.\text{nb\_instructions}$$

Et quant au nombre de cycles de gel dû aux WRITE :

$$\text{write\_freeze\_cycles} = \text{write\_cost} \times \text{write\_rate} \times \text{nb\_instructions} = 10^0 \times 10^{-1} \times \text{nb\_instructions} = 10^{-1}.\text{nb\_instructions}$$

Ainsi, la dégradation des performances est surtout causée par le coût des écritures.